UNIVERSIDADE
DE VIGO

ESCOLA SUPERIOR DE ENXEÑARÍA INFORMÁTICA

Manual Técnico do Traballo de Fin de Grao que presenta

D. Julián Jiménez González

para a obtención do Título de Graduado en Enxeñaría Informática

SuperCars: Desarrollo de videojuego para dispositivos móviles

Maio, 2015

Traballo de Fin de Grao Nº: EI 14/15 - 010

Titor/a: Alma María Gómez Rodríguez
Co-titor/a: David Ramos Valcárcel
Área de coñecemento: Linguaxes e Sistemas Informáticos
Departamento: Informática

Manual Técnico

1. Introducción

1.1. Descripción del sistema

SuperCars es un videojuego de carreras en 2D. En él, el jugador debe mover el vehículo a izquierda y derecha para evitar los obstáculos que van apareciendo en la carretera, así como adelantar a los oponentes. Se pretende que la aleatoriedad en la generación de obstáculos y oponentes provoque situaciones distintas en cada partida, así como el incremento de dificultad.

2. Análisis

2.1. Especificación de Requisitos Funcionales

En este apartado se reflejan todos los Requisitos Funcionales que se han establecido al inicio del proyecto o bien se han añadido posteriormente. Éstos reflejan toda la funcionalidad que es imprescindible para considerar el proyecto como finalizado.

La nomenclatura es RF (Requisito Funcional), seguido de un número de identificación único. Ésta nomenclatura puede ser útil para relacionar cualquier parte de la documentación con los requisitos.

# Requisito	Descripción
RF1	Deben poder configurarse varios parámetros de la carrera como vueltas, rivales y dificultad
RF2	El jugador debe poder controlar su vehículo con unos botones que se muestren en pantalla
RF3	La vista del juego debe ser aérea (en 2D)
RF4	Debe haber un *HUD*[1] que muestre al jugador información sobre la carrera, como vueltas o posición
RF5	Debe guardarse un *ranking* de vueltas rápidas
RF6	La dificultad debe influír en el comportamiento de los rivales
RF7	El jugador debe poder pausar la carrera en cualquier momento

2.2. Especificación de Requisitos No Funcionales

Aquí se detallan los Requisitos No Funcionales. Éstos reflejan características del sistema que, si bien su incumplimiento no hace que el sistema deje de funcionar, sí pueden influír en la experiencia del usuario al utilizar la aplicación.

[1] *HUD son las siglas en inglés de Head-Up Display. Es una pantalla transparente que presenta la información al usuario sin que éste tenga que cambiar su punto de vista para verla. El origen proviene de que el usuario pueda ver la información con la cabeza erguida (Head-Up) y mirando al frente, sin necesidad de bajarla.*

La nomenclatura sigue el mismo patrón que en los *Requisitos Funcionales (2.1)*, siendo RNF (Requisitos No Funcionales) seguido de un número de identificación único, además de poseer la misma utilidad.

# Requisito	Descripción
RNF1	La aplicación debe mantener una tasa de *frames per second* superior a 30
RNF2	La tasa de *frames per second* debe ser estable (no variar más de un 25% cada segundo
RNF3	La carrera debe poder contener 40 o más obstáculos simultáneos
RNF4	La aplicación debe funcionar en modo *portrait*[2] para ofrecer suficiente visibilidad al jugador
RNF5	Al iniciar la aplicación, el menú principal debe ser visible en menos de 5 segundos
RNF6	La transición entre escenas no debe durar más de 2 segundos

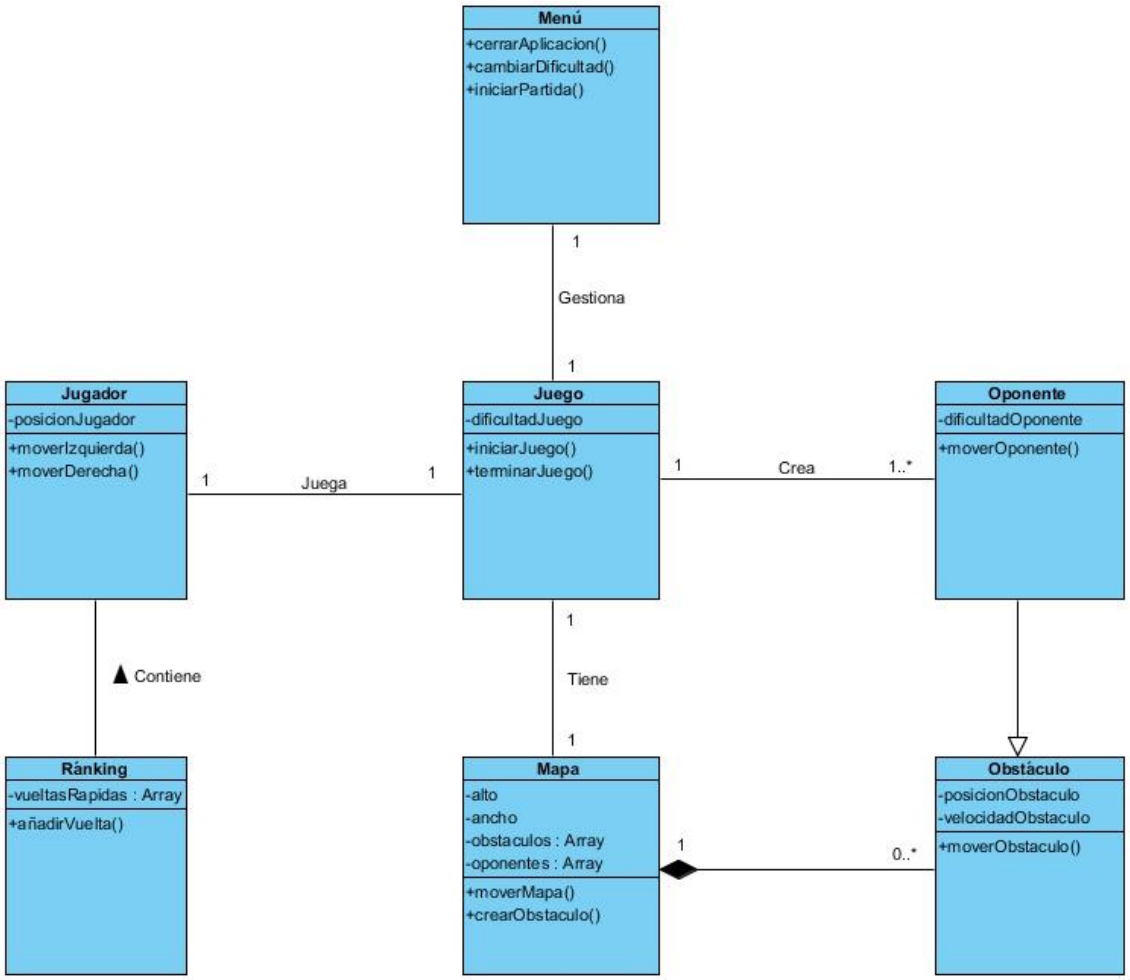
[2] *Portrait hace referencia a la orientación de la aplicación, que debe extender su lado más largo de arriba a abajo de la pantalla (orientación vertical)*

2.3. Modelo de Dominio

En el Modelo de Dominio se reflejan las clases (conceptuales) que se consideran significativas para el dominio del problema.

Estas clases no tienen por qué coincidir con las implementadas finalmente, ya que éstas últimas pueden variar en función del entorno de desarrollo e incluso del de despliegue. La lógica y funcionalidades descritas en estas clases sí deben mantenerse.

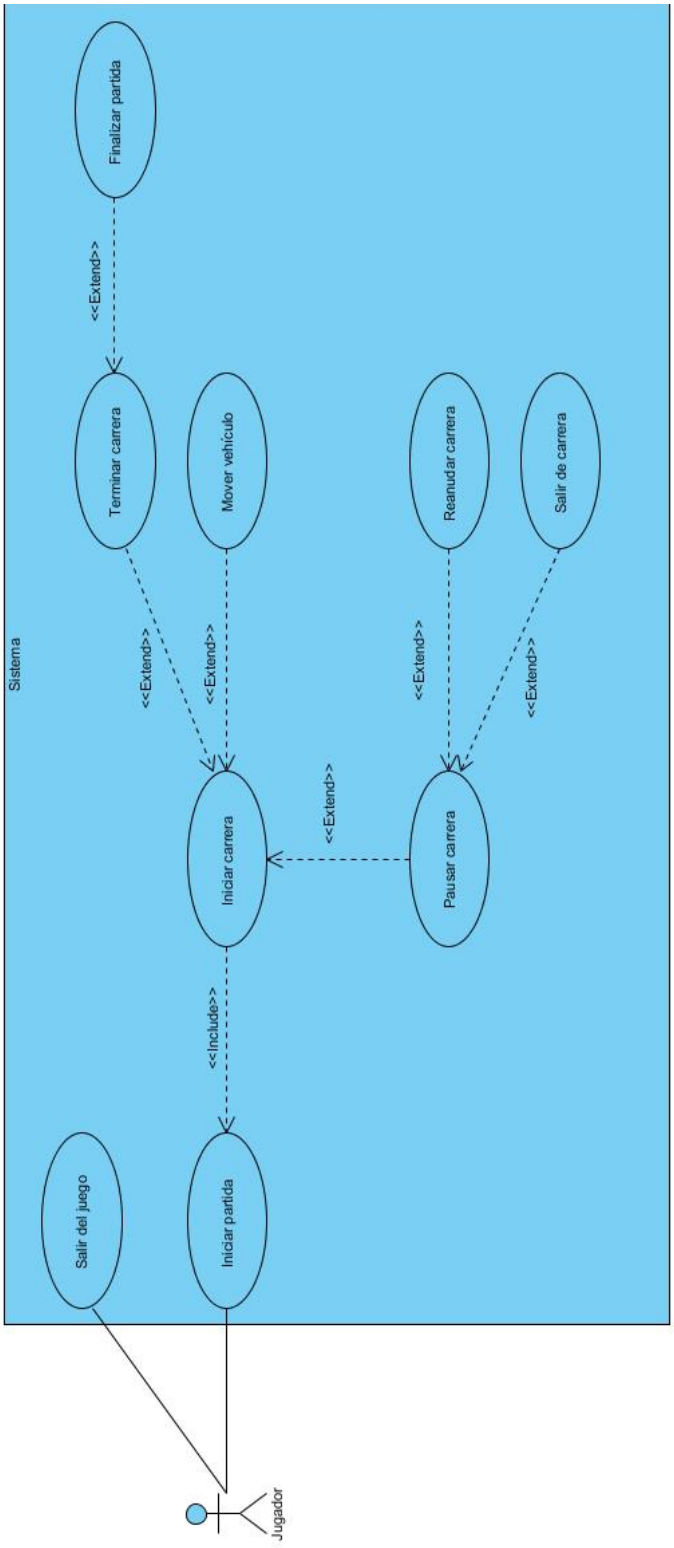
La versión final del Modelo de Dominio que refleja el diseño real de la aplicación se puede encontrar en el apartado *3.1*.



2.4. Descripción de Casos de Uso

2.4.1. Diagrama de Casos de Uso

Este diagrama representa las interacciones del usuario con el sistema, cumpliendo con los requisitos funcionales descritos anteriormente (*2.1*).



A continuación se realiza una descripción detallada de los Casos de Uso, que incluye:

- Breve descripción.
- Precondiciones.
- Postcondiciones.
- Flujo de eventos.
- Flujo alternativo (opcional).

Se obvia el actor debido a que en todos ellos es el jugador.

2.4.2. Descripción detallada de Casos de Uso

2.4.2.1 Iniciar partida

Iniciar partida			
Descripción breve	El jugador pulsa el botón *Play* en *MainMenu* y la escena cambia a *RaceConf*		
Precondiciones	La escena activa es *MainMenu*		
Postcondiciones	La escena activa es *RaceConf*		
Flujo Principal		**Jugador**	**Sistema**
	1	Pulsa el botón *Play*	
	2		Cambia escena activa de *MainMenu* a *RaceConf*

2.4.2.2 Iniciar carrera

Iniciar carrera			
Descripción breve	El jugador configura las opciones de carrera y ésta da comienzo de acuerdo con ellas		
Precondiciones	La escena activa es *RaceConf*		
Postcondiciones	La escena activa es *Race*		
Flujo Principal		**Jugador**	**Sistema**
	1		Muestra parámetros de configuración de la carrera
	2	Ajusta los parámetros de configuración	
	3		Recoge la configuración introducida e inicia la carrera de acuerdo a dicha configuración

2.4.2.3 Mover vehículo

Mover vehículo			
Descripción breve	El jugador pulsa un botón de dirección y el vehículo se mueve al sitio correspondiente		
Precondiciones	La escena activa es *Race*		
Postcondiciones	El vehículo se ha movido a la posición deseada		
Flujo Principal		**Jugador**	**Sistema**
	1	Pulsa botón de movimiento *leftArrow* o *rightArrow*	
	2		Identifica cuál de los botones ha sido pulsado
	3		Mueve el vehículo al lado correspondiente
	4		Se queda a la espera de una nueva interacción
Flujo Alternativo [A3]		**Jugador**	**Sistema**
	1		Si está a la izquierda y se ha pulsado *leftArrow*, no hace nada. Vuelve al paso 4
Flujo alternativo [A3]		**Jugador**	**Sistema**
	1		Si está a la derecha y se ha pulsado *rightArrow*, no hace nada. Vuelve al paso 4

2.4.2.4 Terminar carrera

Terminar carrera			
Descripción breve	El jugador alcanza la línea de meta de la última vuelta y la carrera finaliza		
Precondiciones	La escena activa es *Race*		
Postcondiciones	La escena activa es *EndRace*		
Flujo Principal		**Jugador**	**Sistema**
	1	Alcanza la línea de meta	
	2		Finaliza la carrera
	3		Cambia escena activa de *Race* a *EndRace*

2.4.2.5 Finalizar partida

Finalizar partida			
Descripción breve	El jugador vuelve a *MainMenu* pulsando *back* en la escena *EndRace*, una vez terminada la carrera		
Precondiciones	La escena activa es *EndRace*		
Postcondiciones	La escena activa es *MainMenu*		
Flujo Principal		**Jugador**	**Sistema**
	1	Pulsa *back*	
	2		Cambia escena activa de *EndRace* a *MainMenu*

2.4.2.6 Pausar carrera

Pausar carrera			
Descripción breve	El jugador pulsa el botón *menú* en la escena *Race* y se pausa el juego para mostrar *RaceMenu*		
Precondiciones	La escena activa es *Race*		
Postcondiciones	La escena activa es *RaceMenu*		
Flujo Principal		**Jugador**	**Sistema**
	1	Pulsa botón *menu*	
	2		Pausa escena *Race*
	3		Muestra escena *RaceMenu*

2.4.2.7 Reanudar carrera

Reanudar carrera			
Descripción breve	El jugador pulsa *resume* en *RaceMenu* y se reanuda la carrera		
Precondiciones	La escena activa es *RaceMenu*		
Postcondiciones	La escena activa es *Race*		
Flujo Principal		**Jugador**	**Sistema**
	1	Pulsa *resume*	
	3		Cambia escena activa de *RaceMenu* a *Race*

2.4.2.8 Salir de carrera

Salir de carrera			
Descripción breve	El jugador pulsa el botón *Exit* en *RaceMenu* y la escena cambia a *MainMenu*		
Precondiciones	La escena activa es *RaceMenu*		
Postcondiciones	La escena activa es *MainMenu*		
Flujo Principal		**Jugador**	**Sistema**
	1	Pulsa el botón *exit*	
	2		Finaliza la escena *Race*
	3		Cambia escena activa de *RaceMenu* a *MainMenu*

2.4.2.9 Salir del juego

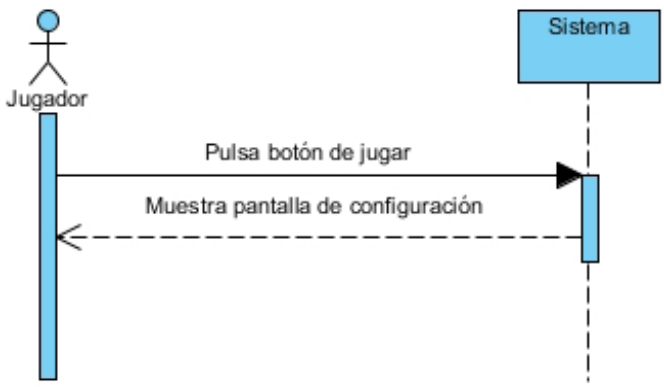
Salir del juego			
Descripción breve	El jugador pulsa el botón *exit* de *MainMenu* y la aplicación se cierra		
Precondiciones	La escena activa es *MainMenu*		
Postcondiciones	Aplicación cerrada correctamente		
Flujo Principal		**Jugador**	**Sistema**
	1	Pulsa *exit* en *MainMenu*	
	2		Cierra la aplicación

2.5. Diagramas de Secuencia

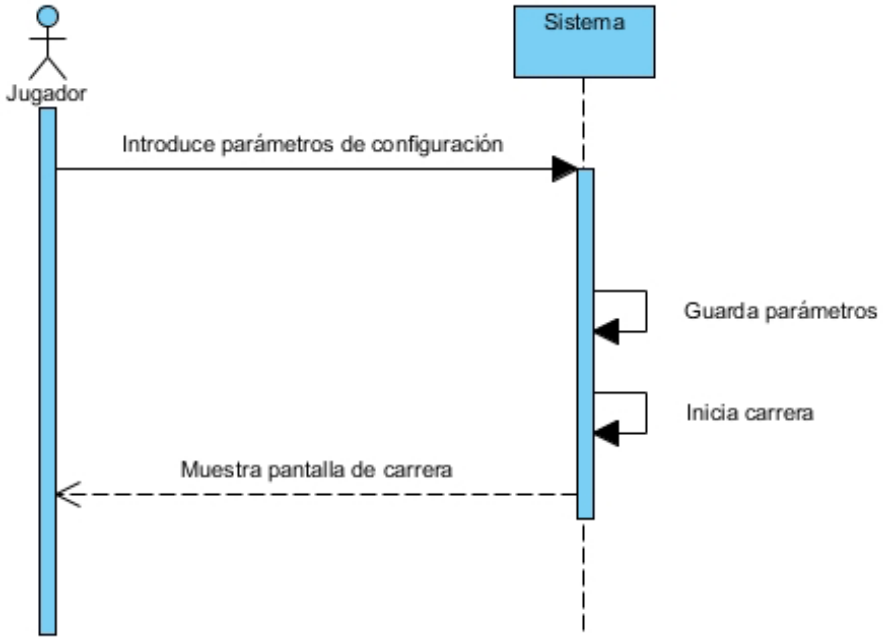
Aquí se muestran los diagramas de secuencia, que reflejan el flujo de información que se produce entre el usuario y el sistema, según las clases y casos de uso de análisis (*2.4.2*).

Ésto es ampliado con los *Diagramas de Secuencia del Sistema (3.3)* que reflejan también el flujo interno del sistema.

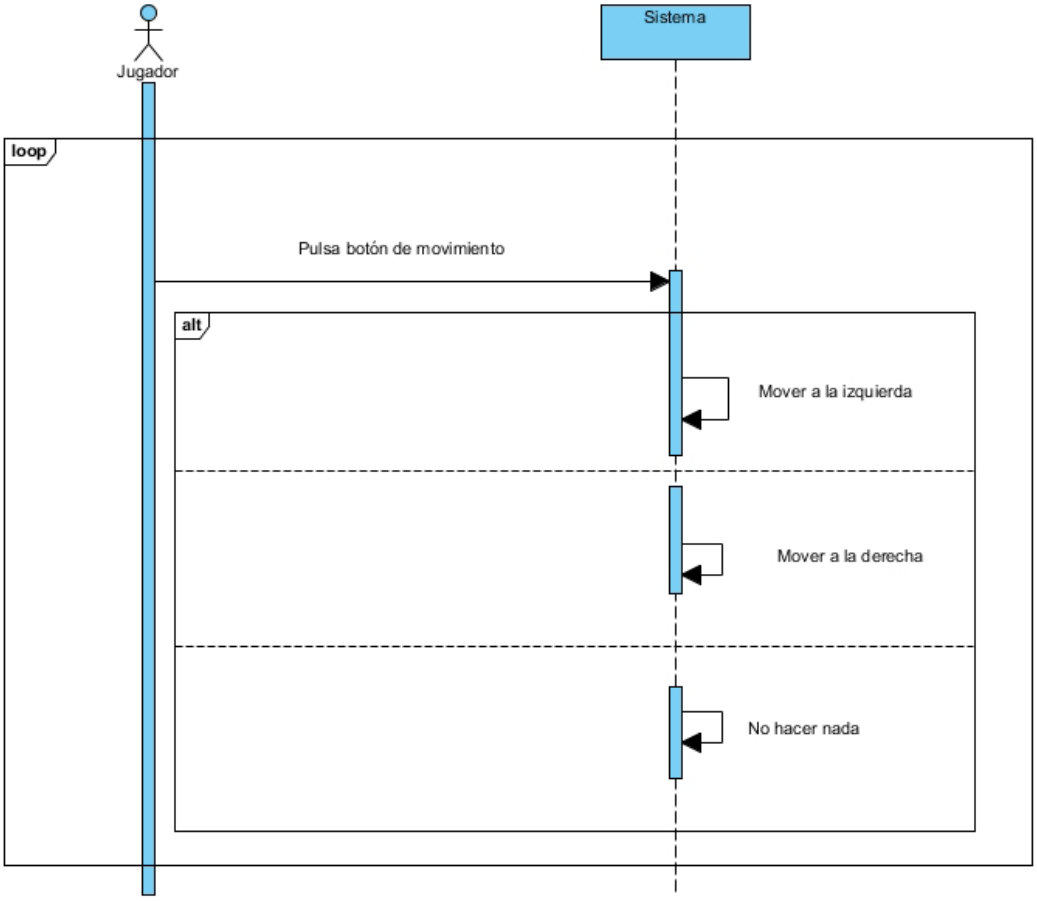
2.5.1. Iniciar partida



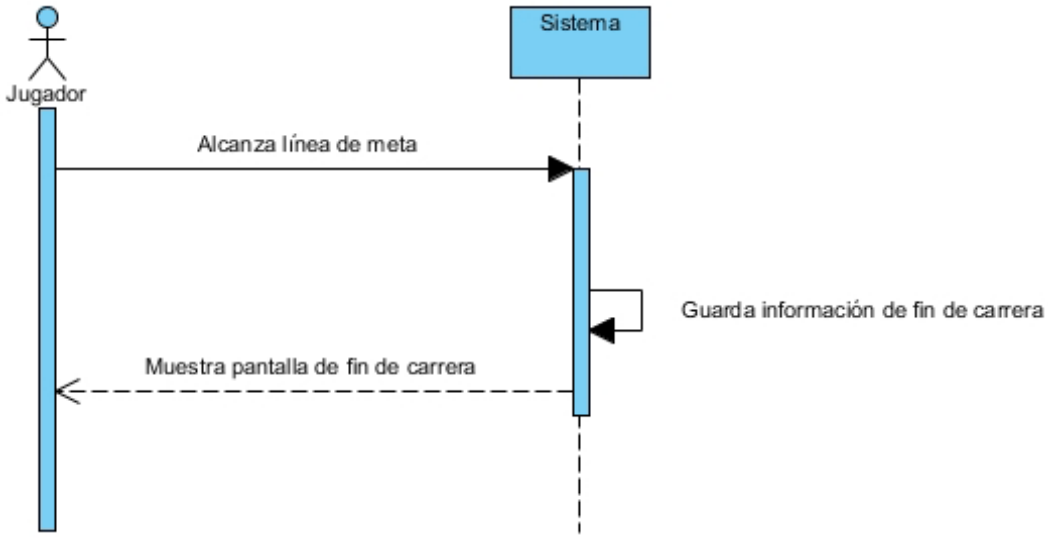
2.5.2. Iniciar carrera



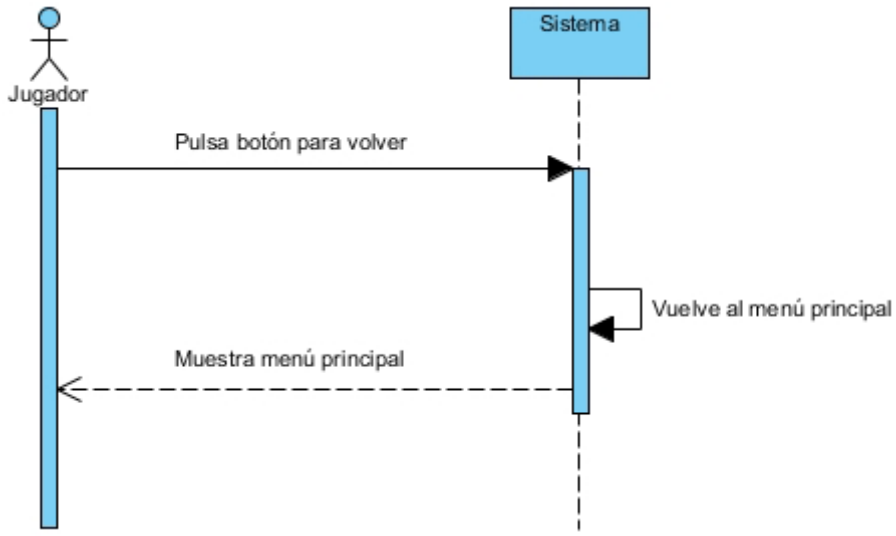
2.5.3. Mover vehículo



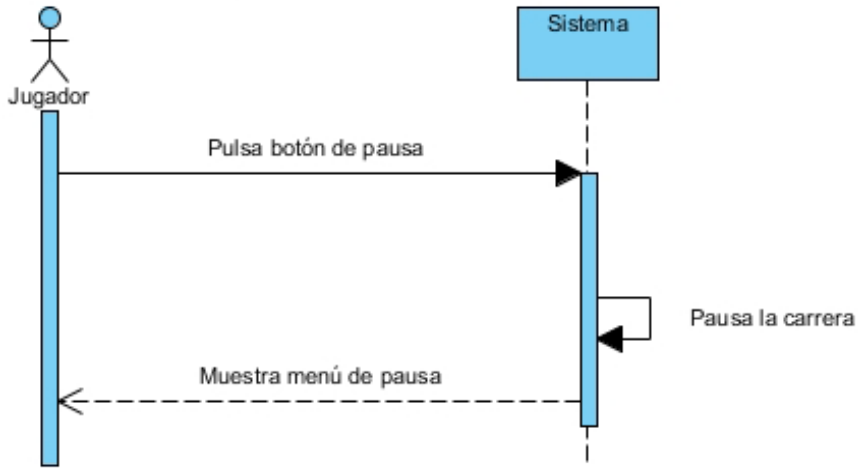
2.5.4. Terminar carrera



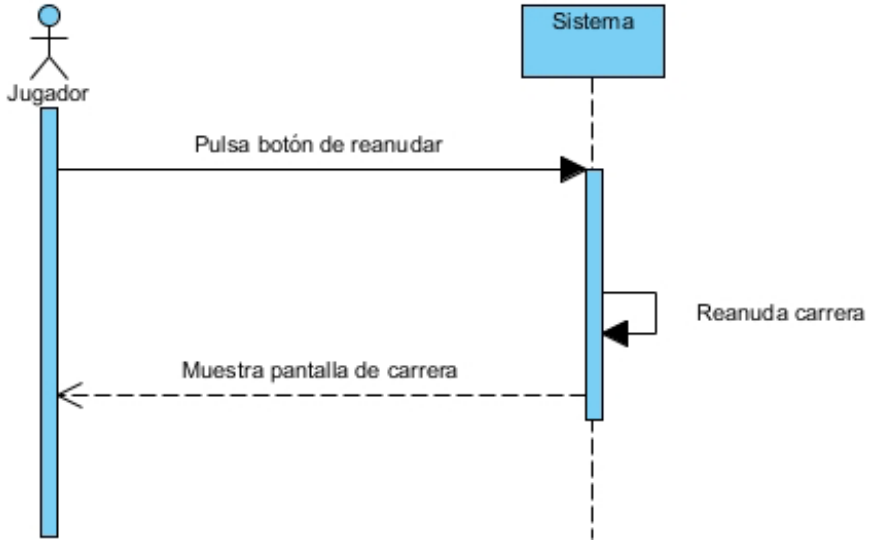
2.5.5. Finalizar partida



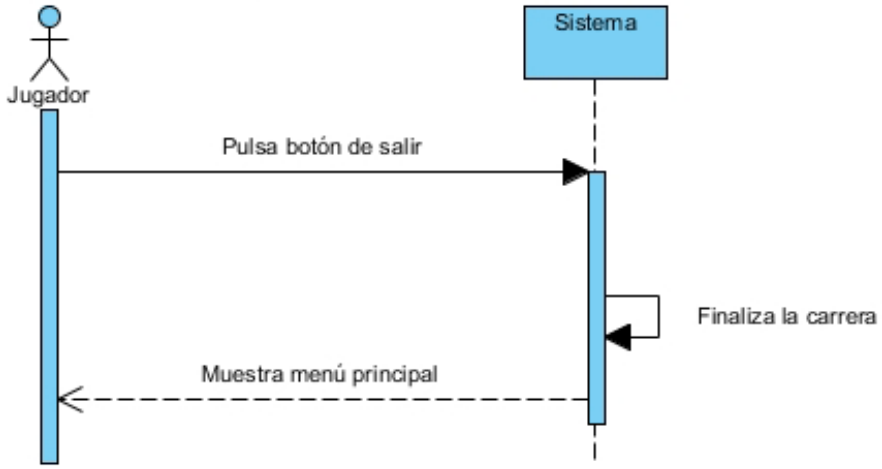
2.5.6. Pausar carrera



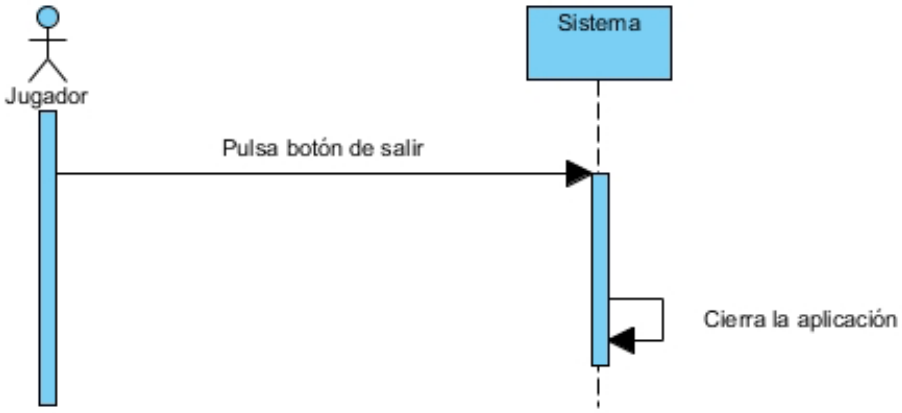
2.5.7. Reanudar carrera



2.5.8. Salir de carrera



2.5.9. Salir del juego



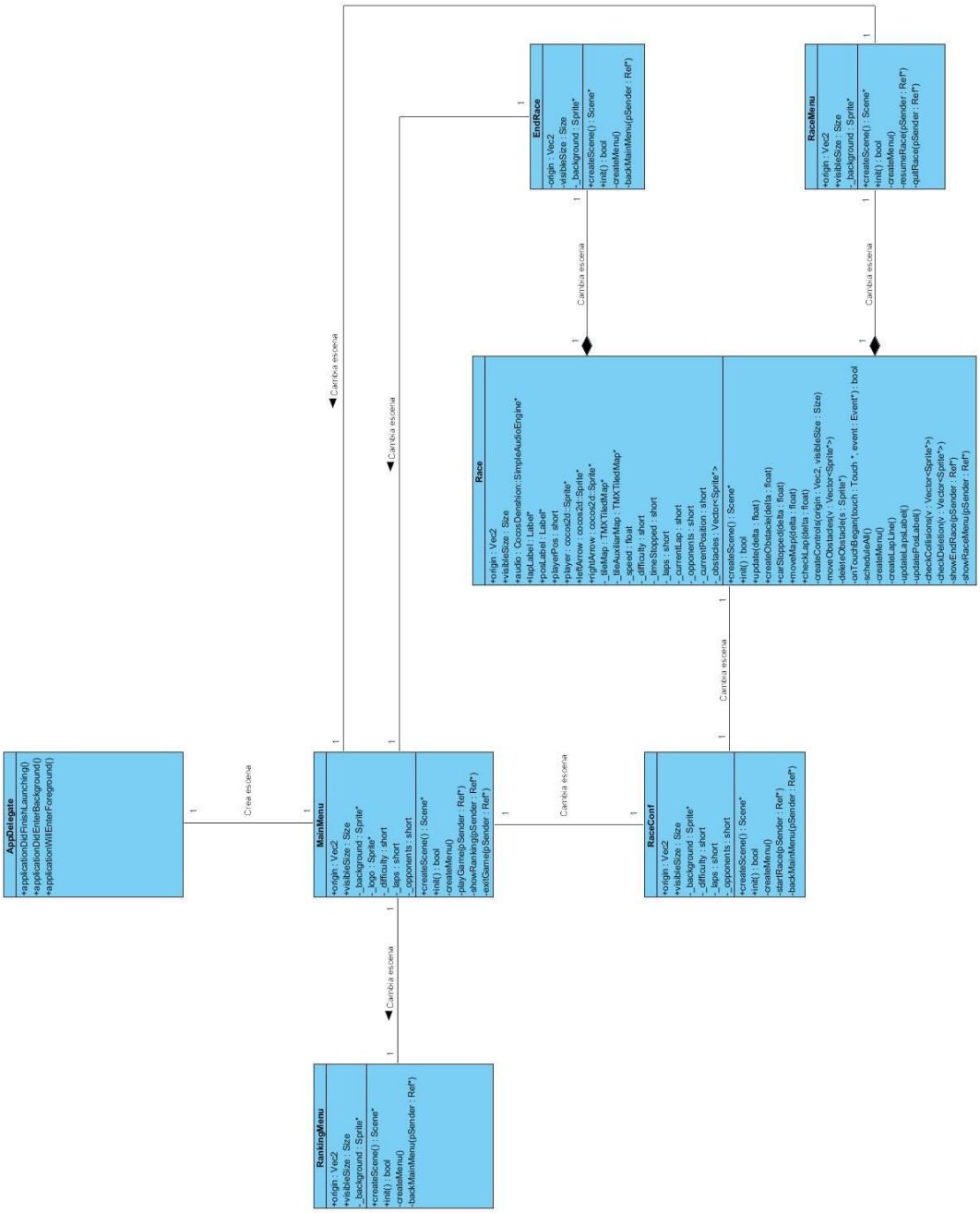
3. Diseño

Las variaciones en el diseño con respecto al *Análisis (2)* están producidas por la adecuación del desarrollo al *framework* Cocos2d-x.

A continuación se muestra el diagrama de clases total, seguido de un diagrama parcial de cada clase.

3.1. Diagrama de clases total

Este es el diagrama de clases total que finalmente refleja el diseño de la aplicación.
Es diferente del Modelo de Dominio *(2.3)* de la fase de *Análisis* pero mantiene la funcionalidad descrita.



Entre las diferencias más importantes de éste diagrama de clases final con el *Modelo de Dominio (2.3)* de la fase de *Análisis*, cabe destacar:

- La entidad Jugador no es necesario representarla como una clase separada, ya que finalmente es una variable de la escena *Race (3.2.5)* y la clase *Sprite* propia de Cocos2d-x es suficiente para implementar la funcionalidad deseada. Sucede lo mismo con las clases *Mapa*, *Obstáculo* y *Oponente*. En el caso de éstos últimos incluso son tratados en un mismo vector, diferenciados por un *Tag*[3]

[3] *Cocos2d-x utiliza Tags en sus clases como un identificador que puede ser único o compartido por distintos elementos.*

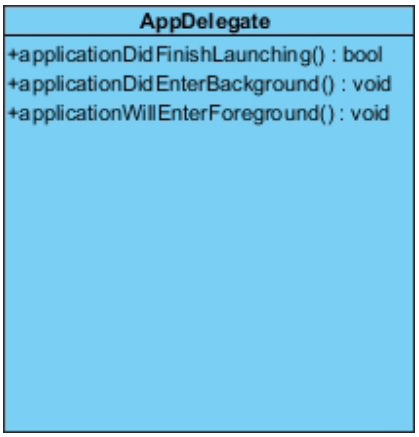
3.2. Diagramas de clases parciales

En este apartado se detallan las clases parciales del sistema, que se pueden identificar con una escena del juego (exceptuando *AppDelegate*).
Todas las clases tienen su diagrama parcial, una breve explicación y sus métodos y variables más importantes.

Existen ciertas variables y métodos que son comunes a casi todas las clases, por lo que se detallan aquí:

- *origin*: variable de tipo *Vec2* que contiene las coordenadas X e Y del origen.
- *visibleSize*: variable de tipo *Size* que contiene la altura y el ancho de la pantalla.
- *createScene()*: es el constructor de la escena, por lo que devuelve un objeto escena básico.
- *init()*: función que se ejecuta siempre y contiene las llamadas principales de la clase. Se puede considerar como una función *main*[4], por lo que el resto de métodos son llamados (directa o indirectamente) desde éste.
- *createMenu()*: crea y coloca los botones del menú correspondiente a esa escena.
- *onTouchBegan(Touch* touch, Event* event)*: se asocia al listener de los botones para que se ejecute cuando el jugador los toca.

3.2.1. AppDelegate

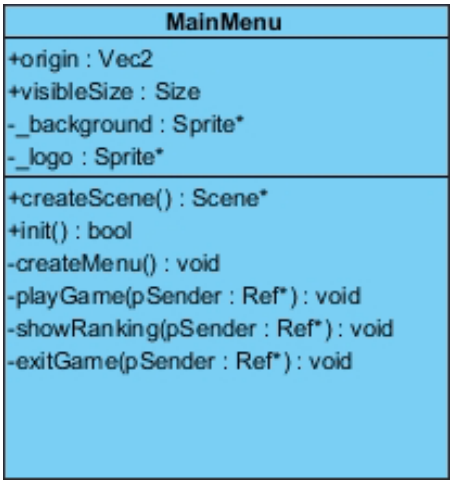


Esta es una clase base propia de Cocos2d-x. Su función es la de gestionar el inicio y fin de la aplicación.

- *applicationDidFinishLaunching()*: se ejecuta cuando la aplicación termina de abrirse.
- *applicationDidEnterBackground()*: se ejecuta cuando la aplicación pasa a segundo plano.
- *applicationWillEnterForeground()*: se ejecuta cuando la aplicación vuelve activarse tras *applicationDidEnterBackground()*.

[4] *Función típicamente llamada al arrancar un programa*

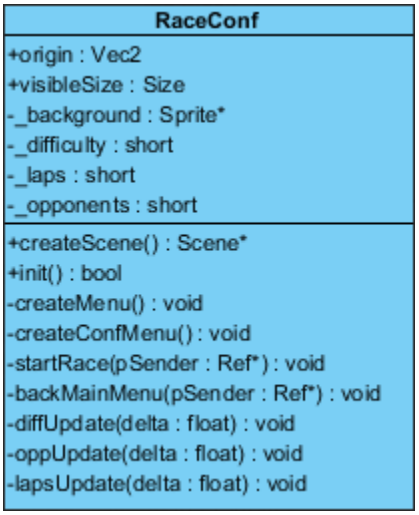
3.2.2. MainMenu



Esta clase se corresponde con el menú principal del juego. Es por tanto la escena creada desde *AppDelegate*.

- *playGame(Ref* pSender)*: crea una nueva instancia de *RaceConf* y reemplaza a ésta.
- *showRanking(Ref* pSender)*: crea una nueva instancia de *RankingMenu* y reemplaza a ésta.
- *exitGame(Ref* pSender)*: finaliza la aplicación.

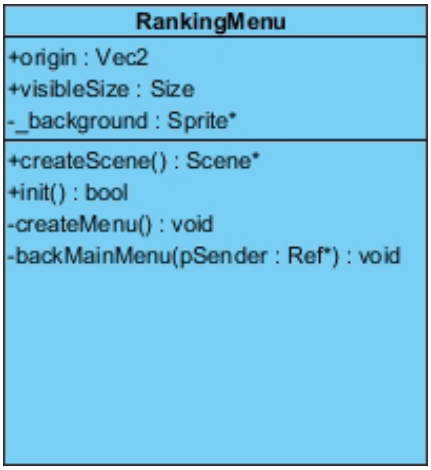
3.2.3. RaceConf



Esta clase corresponde a la escena de configuración de la carrera.

- *startRace(Ref* pSender)*: recoge y guarda los parámetros introducidos, después crea una nueva instancia de *RaceScene* y reemplaza a ésta.
- *backMainMenu(Ref* pSender)*: vuelve a *MainMenu*.
- *diffUpdate(float delta)*, *oppUpdate(float delta)*, *lapsUpdate(float delta)*: se emplean para actualizar en tiempo real el valor de los *slider*.

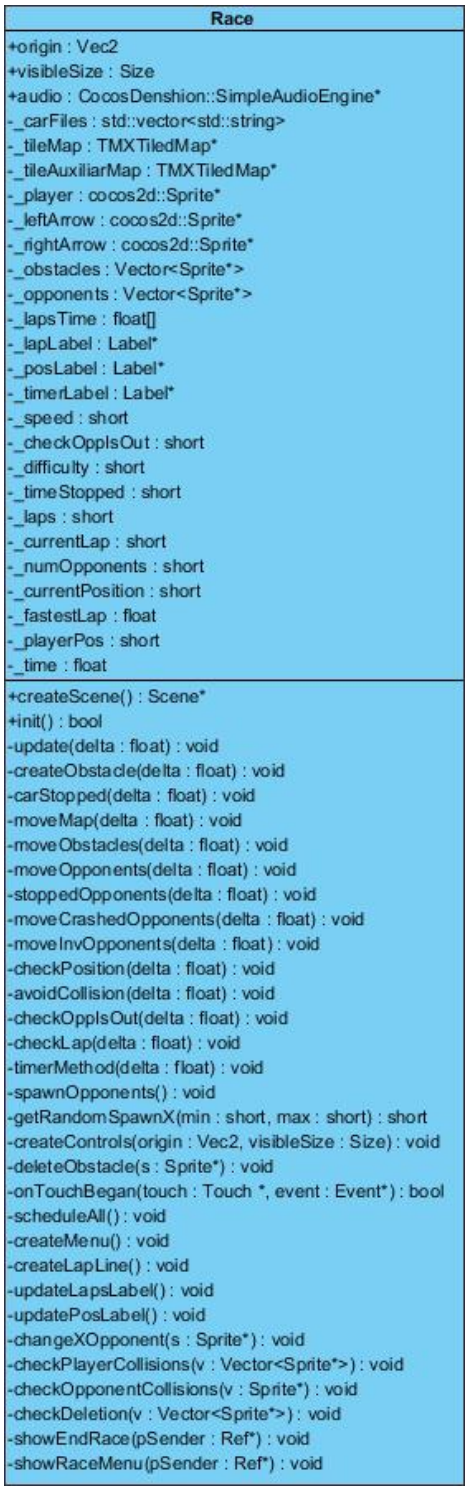
3.2.4. RankingMenu



Esta clase corresponde a la escena de visualización del *ranking* de vueltas rápidas.

- *backMainMenu(Ref* pSender)*: vuelve a *MainMenu*.

3.2.5. Race

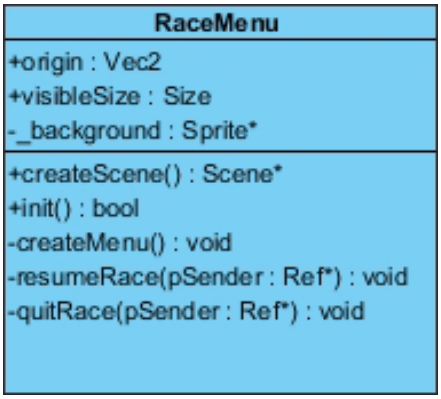


Esta clase es la escena principal del juego. Contiene todo lo visible en la escena de carrera. Tan sólo se detallan los métodos y variables más relevantes.

- *_carFiles*: es un vector de *strings* que guardan el nombre de los archivos de textura de los vehículos.

- *_difficulty*, *_laps* y *_numOpponents*: estas variables se recogen de la información introducida en *RaceConf*.
- *_speed*: esta variable controla cómo de rápido se mueve el mapa y los rivales. Se calcula a partir de *_difficulty* y *_opponents*.
- *scheduleAll()*: este método programa (*schedule)* los métodos que deben ejecutarse cada *frame* o cada tiempo estimado.
- *update(float delta)*: bucle principal del juego. Se ejecuta cada *frame* y *delta* es el tiempo transcurrido entre el *frame* actual y el anterior (este tiempo es variable). Desde este método se realizan llamadas a *checkPlayerCollisions(_obstacles)* y *checkDeletion(_obstacles)*.
- *carStopped(float delta)*: éste método pasa a ser el bucle principal cuando el vehículo del jugador se para (ha colisionado con un obstáculo). Tras un tiempo llama de nuevo a *scheduleAll()* y se desactiva. *moveCrashedOpponents(float delta)* realiza una tarea similar para los oponentes.
- *moveMap(float delta)*: mueve el mapa en función de *_speed*.
- *moveObstacles(Vector<Sprite*> v)*: mueve los obstáculos en función de *_speed* para generar la sensación de que éstos están fijos en el mapa.
- *createObstacle(float delta)*: crea un obstáculo cada *delta* segundos en una posición semi-aleatoria.
- *checkLap(float delta)*: comprueba cada *delta* segundos si la vuelta actual es igual al número máximo de vueltas, para terminar la carrera.
- *checkOppIsOut(float delta)*: comprueba cada *delta* segundos si algún oponente está fuera de la carretera y, en caso afirmativo, lo devuelve al centro de la misma.
- *getRandomSpawnX(short min, short max)*: devuelve una posición en el eje de coordenadas X que se encuentre entre los límites de la carretera.

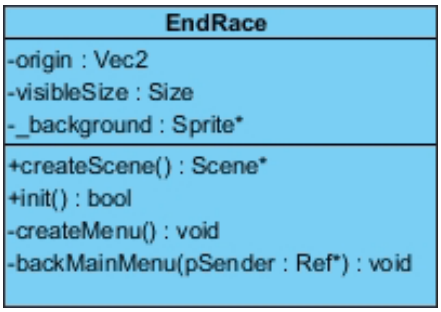
3.2.6. RaceMenu



Esta escena es el menú llamado desde *Race*.

- *resumeRace(Ref* pSender)*: reanuda la carrera.
- *quitRace(Ref* pSender)*: finaliza la carrera, crea una nueva instancia de *MainMenu* y se reemplaza por ésta.

3.2.7. EndRace



Esta es la escena que sigue a *Race* cuando la carrera termina. Muestra el tiempo obtenido y permite volver a *MainMenu*.

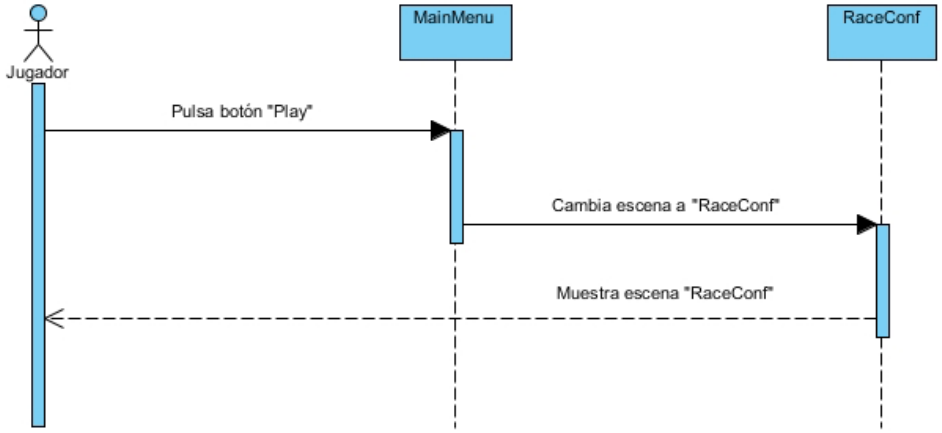
- *backMainMenu(Ref* pSender)*: devuelve a *MainMenu*.

3.3. Diagramas de Secuencia del Sistema

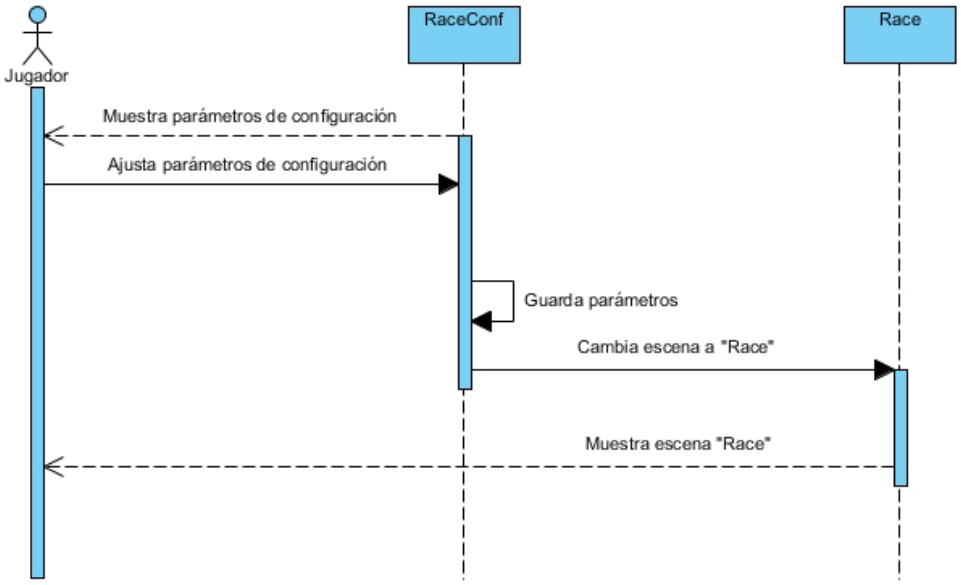
En estos diagramas se pueden ver las transiciones de información y los cambios que se producen en el sistema debido a la interacción del Jugador.

Por motivos de legibilidad, se ha decidido describir las relaciones con frases cortas en lugar de indicar el nombre del método correspondiente, aunque cada relación entre clases del sistema se corresponde con la llamada a algún método.

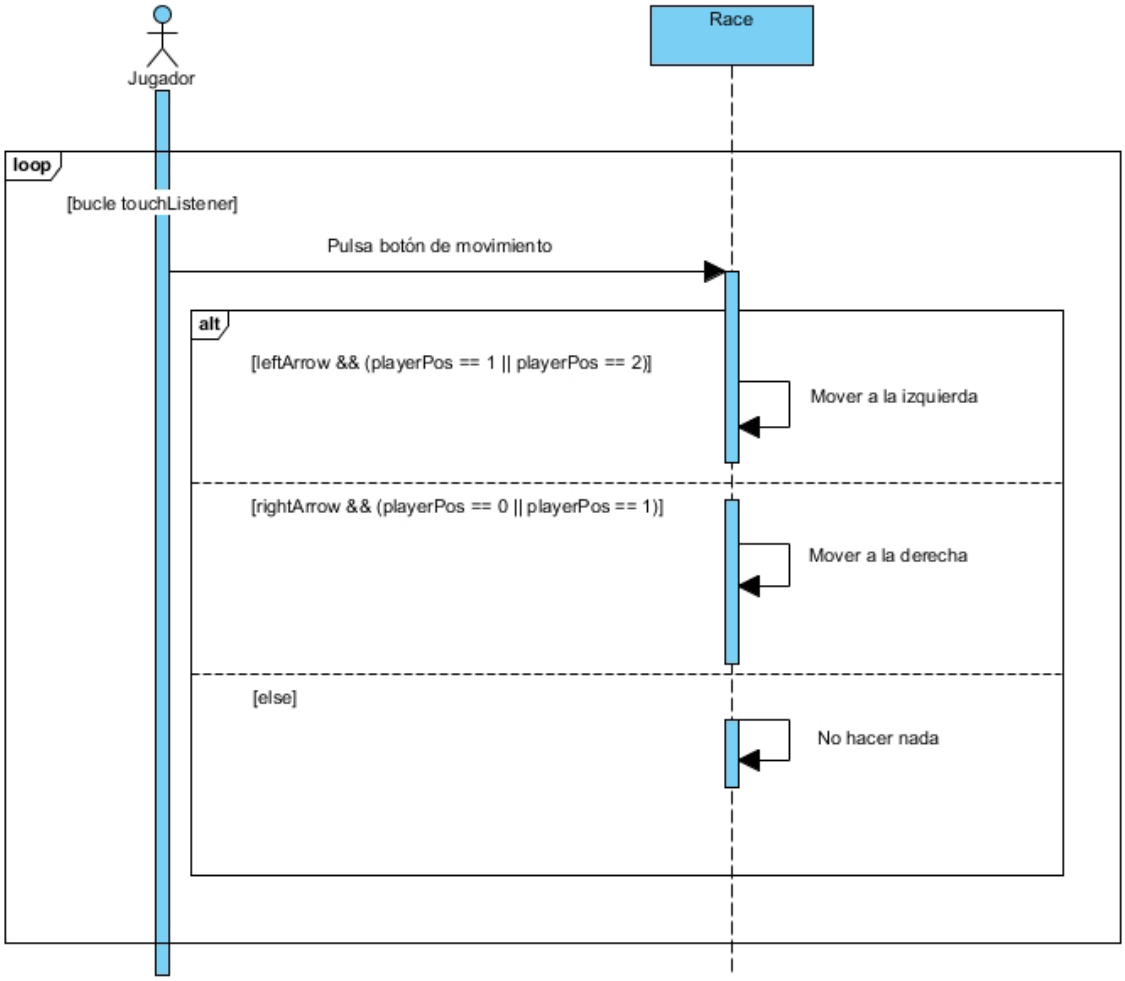
3.3.1. Iniciar partida



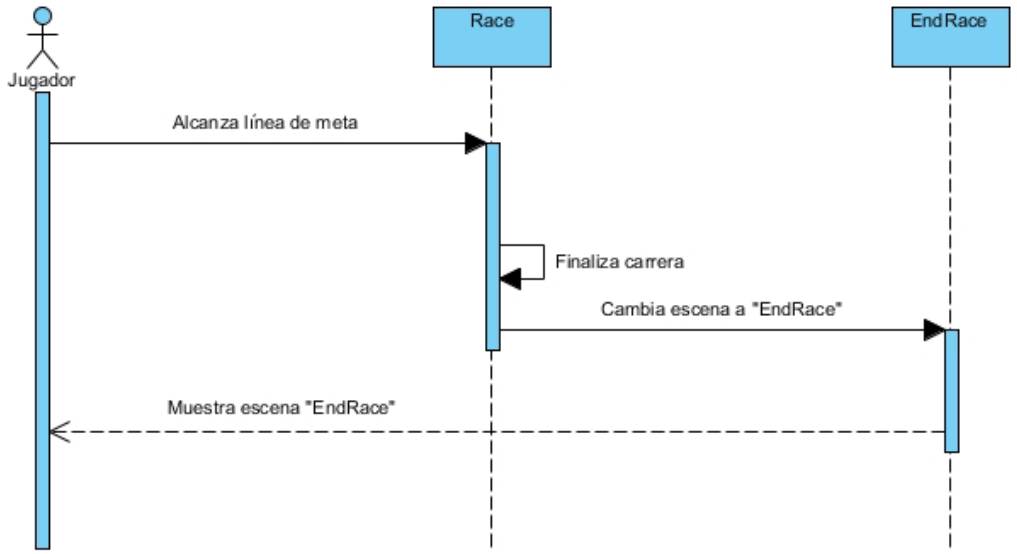
3.3.2. Iniciar carrera



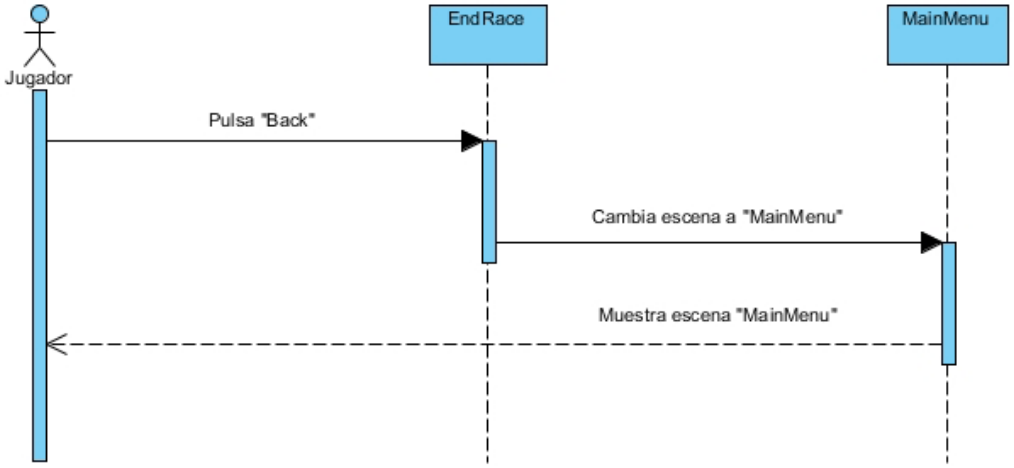
3.3.3. Mover vehículo



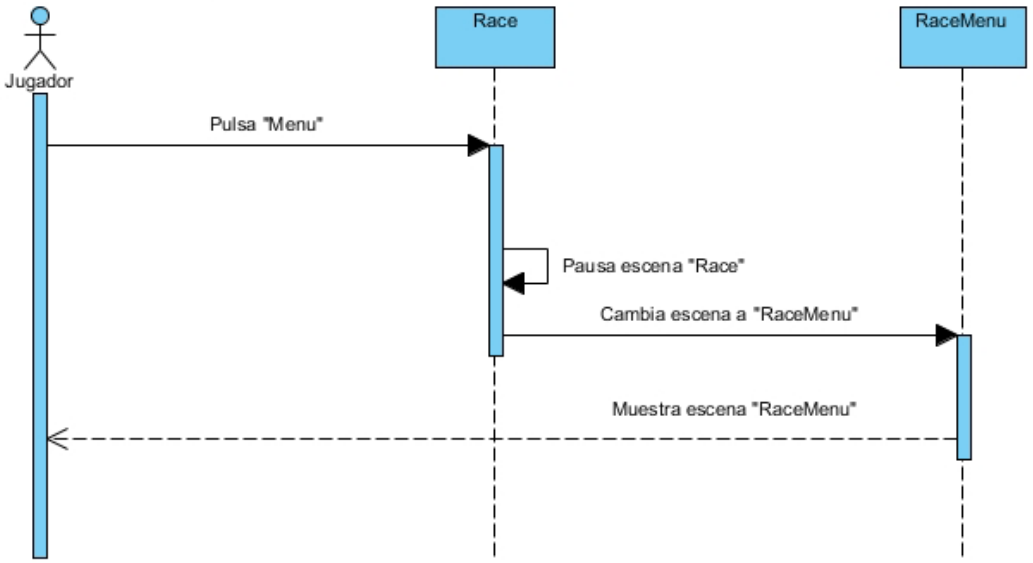
3.3.4. Terminar Carrera



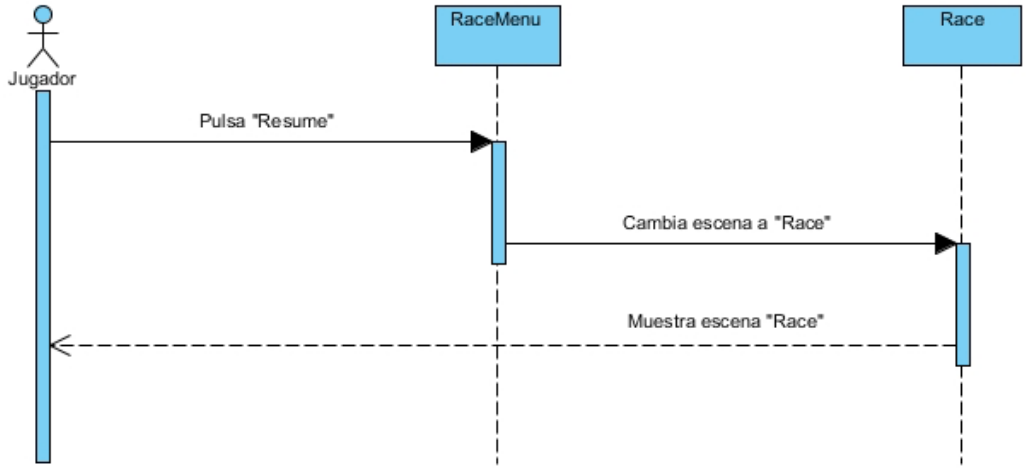
3.3.5. Finalizar partida



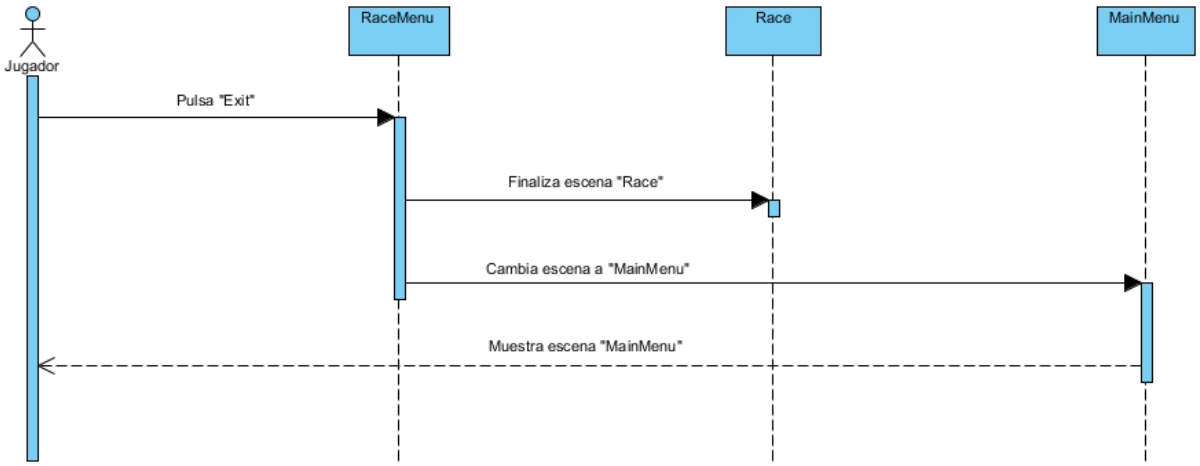
3.3.6. Pausar carrera



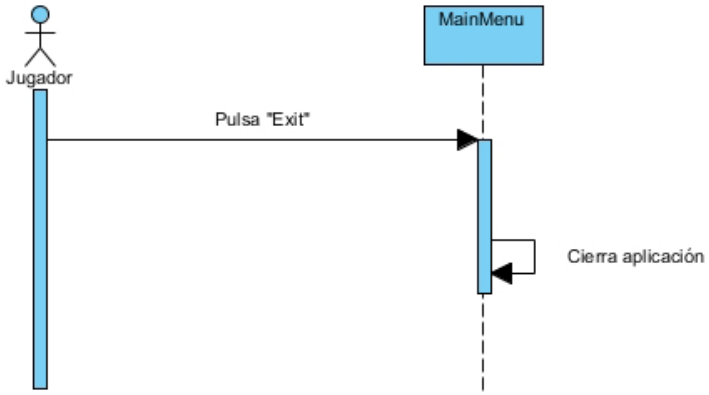
3.3.7. Reanudar Carrera



3.3.8. Salir de carrera



3.3.9. Salir del juego



4. Detalles de implementación

Aquí se explican brevemente los detalles de implementación que se considera que son importantes, bien por ser características muy específicas de Cocos2d-x o del tipo de proyecto.

4.1. Sprites

El jugador, los rivales, los obstáculos… Todos ellos son *Sprites*. Un *Sprite*, en 2D, es una imagen que se manipula modificando sus propiedades de posición, rotación, escala u opacidad, entre otras. Por lo tanto, cualquier imagen que sufra transformaciones en la pantalla, puede ser un *Sprite*.

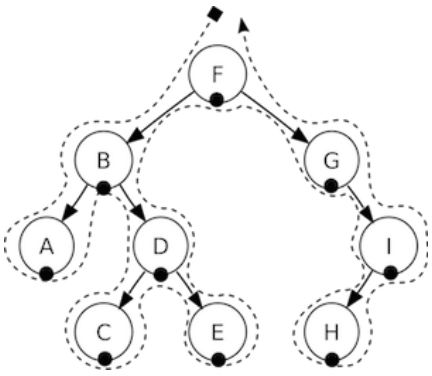
Para crear un Sprite en Cocos2d-x, tan sólo se necesita 1 línea:

```
_player = Sprite::create("audi_r8.png");
```

Tras crearlo, hay que añadirlo al árbol de nodos de la escena, de la siguiente forma:

```
this->addChild(_player, 100);
```

Donde, el segundo parámetro, es el orden de prioridad (*Z-order*) de la propia escena. Es un parámetro opcional, pero es recomendable gestionarlo ya que, habiendo dos *Sprites* en la misma posición, el que mayor *Z-order* tiene se muestra por encima. Esto se debe al orden de renderizado de la escena, que se puede ver en el *scene graph*[5]:



(Imagen tomada de la documentación oficial de Cocos2d-x)

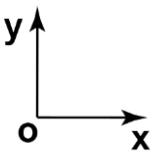
El *Renderer*[6] recorre el árbol empezando por la izquierda y hacia abajo, después la raíz y después la parte derecha, de forma recursiva. Así, si suponemos que *B* y *G* son dos elementos en la misma posición, *G* se superpondría a *B*, ya que se renderiza después. Esto también es aplicable a otro tipo de nodos, que también son añadidos al árbol de la escena. Por defecto, al añadir un Sprite, su *Z-order* es superior al de los elementos añadidos anteriormente.

[5] *El scene graph es representado como un árbol y contiene todos los nodos pertenecientes a la escena*
[6] *Sistema de Cocos2d-x que se encarga de renderizar la escena*

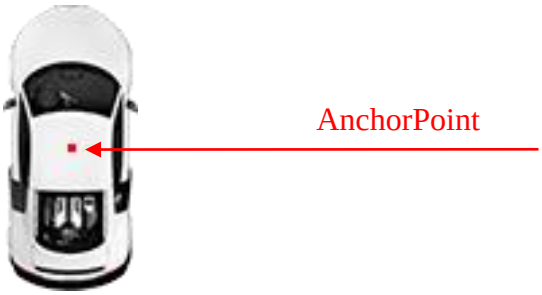
Se presupone que, en el directorio *Resources/* del proyecto, existe un archivo *.png* llamado *audi_r8.png*. Lo que se obtendría, sería algo como esto:

Nótese que el fondo, en este caso, tan sólo refleja la ausencia de otros elementos, siendo negro el color por defecto. En el caso del *Sprite* que se ha añadido, tiene transparencia ya que es un *.png*, algo interesante para visualizar formas que no sean rectangulares.

A continuación, es habitual configurar el *AnchorPoint*. Esto es, un punto (dentro del Sprite) sobre el que se aplican las transformaciones y cuyo valor por defecto es (0.5, 0.5). Este valor por defecto se correspondería con el centro, siendo 1 el valor máximo en ambas coordenadas y 0 el mínimo. El eje de coordenadas que utiliza Cocos2d-x es el mismo que utiliza OpenGL, el cual se rige por la "Regla de la Mano Derecha" (Right Handed Cartesian Coordinate System).



Por lo tanto, el *AnchorPoint* por defecto sería el siguiente:



Para cambiarlo, tan sólo hay que pasar como parámetro un *Vec2*, un tipo de dato de Cocos2d que representa una coordenada en 2D. Se haría de esta forma:

```
_player->setAnchorPoint(Vec2(0.5, 1));
```

Así, habremos modificado el *AnchorPoint* en su eje Y:

Es importante prestar atención al *AnchorPoint* de un *Sprite* porque, sino, es probable que éste sea mal posicionado en algún momento.

La modificación de otras propiedades como *scale*, *position*, *skew* o *rotation*, se realiza de forma análoga, teniendo siempre en cuenta el *AnchorPoint*.

Habitualmente, los *Sprites* se utilizan para representar objetos que deben moverse por la pantalla. Para esto, se pueden crear *Actions*, que después uno o varios *Sprites* ejecutan. En todos los juegos se utilizan *Actions* de alguna forma, ya que es lo que permite modificar los *Sprites* a lo largo del tiempo. Algunas de las más habituales son: *MoveBy* y *MoveTo*. La diferencia entre estas dos reside en que, mientras *MoveBy* indica en qué dirección mover el nodo a lo largo de un tiempo *t*, con *MoveTo* se indica a qué posición concreta debe moverse el nodo a lo largo de ese tiempo.

Para crear una acción, es necesaria una sentencia como la siguiente:

```
auto moveRight = MoveBy::create(0.25, Vec2(100, 0));
```

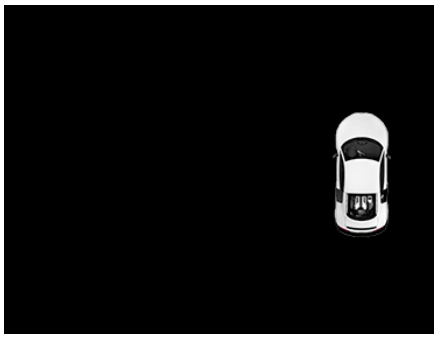
Con la palabra reservada *auto*, se evita tener que especificar el tipo de la variable. En este caso, se indica que, el *Sprite* que ejecute esta acción, debe moverse 100px[7] en el eje X, durante 0.25s. Si el *Sprite* está en la posición (50, 50), tendría que aumentar su coordenada X en 100 (hasta llegar a 150), a lo largo de 0.25s.

Para que el *Sprite* que se creó antes, *_player*, ejecute esta *Action*, tan sólo hay que añadir:

```
_player->runAction(moveRight);
```

[7] *px es la nomenclatura habitual para "Pixel"*

Por lo tanto, suponiendo que esto se añade inmediatamente después de la creación del *Sprite*, obtendríamos el siguiente resultado:



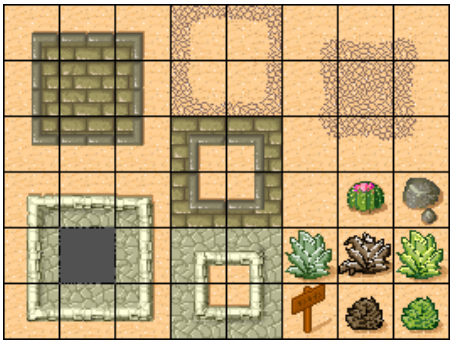
Nótese que la proporción de píxeles no se ha tenido en cuenta.

4.2. Mapa TMX

La carretera y la ornamentación que se pueden ver durante la carrera, constituyen el mapa del juego. Habitualmente, en 2D se utilizan *TileMaps* para representar el mapa, ya que facilita ciertas tareas y permite hacer más cosas que si fueran imágenes planas. En el caso de este proyecto se ha empleado Tiled como herramienta para crear y modificar el *TileMap*, guardando el archivo *.tmx* en formato XML.

Como su propio nombre indica, los *TileMaps* emplean *tiles*, que son como "baldosas" que pueden tratarse de forma independiente, además de otras facilidades como convertirse en *tiles* de colisión. Los *tiles* pueden, también, separarse en capas, lo que permite tanto tener más organizados los grupos de *tiles*, como superponer unos a otros. Más adelante habrá un ejemplo de esto.

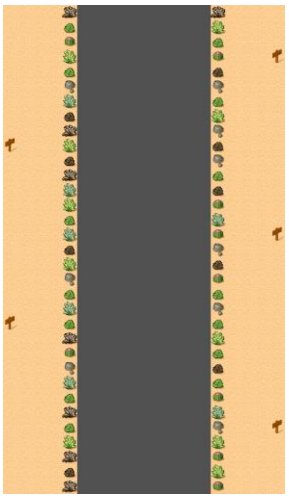
Para crear un *TileMap*, se necesita una imagen en la que se encuentren todos los *tiles* que se vayan a utilizar. En cierto modo, lo que se hace es apilarlos en una misma imagen en lugar de tener una imagen separada para cada *tile*. Esta es una técnica habitual incluso con texturas en 3D, ya que permite reducir el espacio necesario en memoria. Así pues, la imagen empleada en *SuperCars* es la siguiente[8]:



[8] *La plantilla es propia de Tiled, con la modificación del tile empleado para la carretera*

Se pueden identificar claramente los *tiles*, separados por una línea negra de 1px. Para que no haya problemas a la hora de añadirlos al mapa, al abrir el archivo como conjunto de patrones, se tiene que especificar esta separación, así como el tamaño en píxeles de cada *tile*. Empleando Tiled se puede "pintar" un mapa y, como ya se ha mencionado, es recomendable separar los *tiles* en grupos. En el caso de *SuperCars*, se han creado 3 capas: *background*, *road* y *decoration*.

Superponiendo estas capas, se puede exportar como imagen o como *.tmx*, para obtener algo como esto:



La razón de utilizar estas 3 capas, y no otras, es principalmente la organización. Además, en caso de eliminar la capa *decoration*, siempre habría un fondo (*background*) y no se quedaría ningún *tile* vacío.

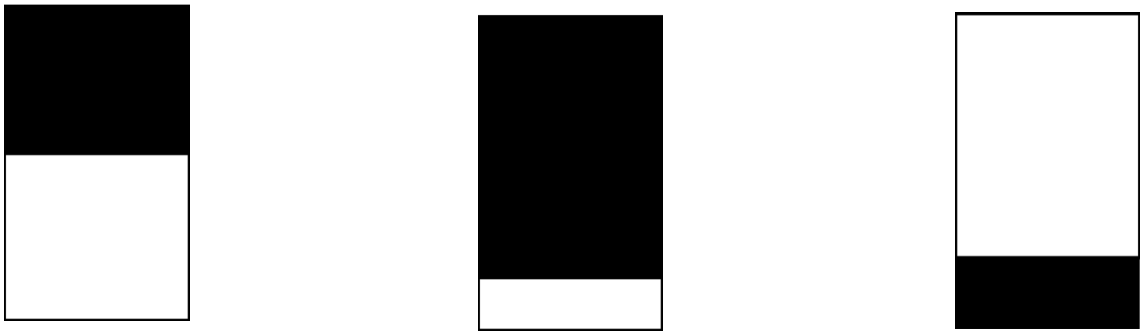
Para crear el mapa en Cocos2d-x, una vez obtenido el archivo *.tmx*, tan sólo hay que añadir una línea similar a esta:

```
_tileMap = TMXTiledMap::create("background.tmx");
```

A partir de aquí, el mapa puede ser manipulado del mismo modo que otros nodos, como los *Sprites* explicados anteriormente (*4.1*). También se puede jugar con la opacidad y situar un mapa sobre otro, si es necesario.

En *SuperCars*, era necesario crear la sensación de que el mapa era "infinito". De esta forma, tanto si la carrera es de 3 vueltas como si es de 15, el código sería el mismo. Para ello, lo que se hace es crear 2 mapas idénticos y comprobar, cada *frame* (más detalles sobre esto en *4.3* Métodos con ejecución programada), cuál de ellos está fuera de la pantalla, para situarlo a continuación del otro.

A continuación se muestra un ejemplo de la transición que se produciría entre los dos mapas. Supóngase que el cuadrado blanco es el primer mapa y el negro el segundo. Inicialmente, el segundo mapa estaría colocado por encima del primero, fuera de la vista del jugador. Cuando el primer mapa desaparece por abajo, se mueve inmediatamente encima del segundo. Así, cuando el segundo comience a desaparecer, el jugador verá el primer mapa de nuevo. El proceso se repite en bucle mientras dure la partida.



4.3. Métodos con ejecución programada

Tanto en 2D como en 3D, todos los juegos tienen un bucle principal, conocido como *game loop*. Éste se encarga, principalmente, de 3 tareas:

1. Inicializar el juego.
2. Actualizar el estado del juego.
3. Dibujar el juego en pantalla.

En el caso de Cocos2d-x, no existe un método como tal para estas tareas. Cada escena representa una pantalla (generalmente, no tiene por qué ser exactamente así), y cada una tiene varios métodos que gestionan todo lo mencionado.

La inicialización se produce en el método *init()* de cada escena, donde se realizan las llamadas principales, del mismo modo que un método *main()* en otro tipo de programas. Desde aquí deben realizarse todas las tareas de inicialización de variables, configuración, creación de actores de la escena, menús… Éste método devuelve una variable de tipo *bool*, que podría servir para averiguar desde otra escena si ésta se ha iniciado.

Una vez se ha inicializado, se tiene que actualizar cada cierto tiempo el estado del juego, ya sea la posición de los actores, el valor de las variables o la información que se muestra en pantalla. Esto se consigue "programando" (s*cheduling*) métodos, ya sea para que se ejecuten cada cierto tiempo, o cada *frame*. Para programar un método, tan sólo hace falta indicarle a la escena qué método y cada cuánto tiempo lo debe ejecutar.

```
this->schedule(schedule_selector(Race::moveMap), (float) 0);
```

En este caso, se está programando el método *moveMap* para que se ejecute cada 0 segundos, o lo que es lo mismo, cada *frame*. En lugar de ello, se puede poner otro valor (como *float*), siendo *1.f* equivalente a 1 segundo, o dejarlo vacío, en cuyo caso se toma por defecto 0.

El método, por defecto, para esta tarea, se llama *update()*, aunque se pueden añadir todos los métodos que se quiera, teniendo en cuenta que sobrecargar la escena puede provocar una caída en los *fps*.

En la declaración del método, es necesario que reciba como parámetro un *float*, que se puede utilizar dentro del método como el tiempo que ha pasado desde la última ejecución. Así, por ejemplo:

```
bool Race::init() {
...
      this->schedule(schedule_selector(Race::timerMethod),
(float) 0.01);
...
}

void Race::timerMethod(float dt) {
      _time += dt;
      ...
}
```

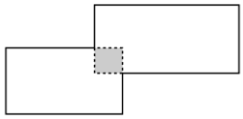
Lo que se está haciendo aquí es, primero, programar un método *timerMethod* para que sea ejecutado cada 0.1 segundos. Éste recibe, como se ha mencionado, una variable *float* en la que se tiene el tiempo pasado desde la última ejecución. Además, utiliza éste tiempo *dt* para aumentar el valor de un contador de tiempo *_time*.

En cuanto a las tareas de dibujado del juego en pantalla, el desarrollador sólo necesita emplear *addChild()* como se mencionó anteriormente, para que el *Renderer* de Cocos2d dibuje los elementos.

4.4. Colisiones

Las colisiones, en los juegos, se producen cuando dos objetos, actores, nodos… entran en contacto. Esto es, en todo momento hay una *Bounding Box*, que es una caja imaginaria que se dibuja alrededor del objeto, teniendo en su interior todo el contenido del objeto. La bounding box del *Sprite* empleado anteriormente (*4.1*), sería la siguiente:

Así pues, cuando las *Bounding Boxes* de 2 objetos comparten 1 o más píxeles, se dice que se está produciendo una colisión. Ésta colisión puede recogerse para actualizar el estado del juego, o ignorarse. En este caso, por lo tanto, se estaría produciendo una colisión:



En este caso, en *SuperCars* se comprueba cada *frame* si existen colisiones entre jugador y obstáculos, jugador y oponentes u oponentes y obstáculos.

4.5. Eventos

Los eventos son una de las partes más importantes de los juegos, ya que gestionan las interacciones del jugador. En el caso de los juegos para dispositivos móviles, los *Touch Events* suelen ser los más importantes, ya que es poco habitual emplear periféricos en ellos.

En Cocos2d-x, *EventDispatch* es el mecanismo que responde a los eventos de usuario, sean del tipo que sean (ratón, teclado, táctiles… hay que recordar que Cocos2d-x se puede utilizar para otras plataformas). Al igual que para los nodos, los eventos también tienen un árbol de prioridad. En éste caso, cada hoja del árbol es un puntero al nodo correspondiente (el nodo para el que ha sido activado el evento).

```
auto touchListener = EventListenerTouchOneByOne::create();
```

Con esta línea, se crea un nuevo *listener*, un elemento que se queda a la espera de recibir eventos. En otras palabras, creamos algo similar a un *trigger*, en este caso del tipo *TouchOneByOne*, que implica que los eventos *Touch* los gestiona de 1 en 1. A mayor prioridad de un *listener*, antes se gestiona éste. Para que el *listener* funcione, hay que activarlo:

```
touchListener->setEnabled(true);
```

Además, por si se pudiese dar el caso de que dos elementos con *listener* asociados se superpusiesen, hay que hacer que se consuma el evento. Esto significa que, el evento, no pasa a otros objetos que estén por debajo.

```
touchListener->setSwallowTouches(true);
```

Una vez se tiene el *listener*, hay que establecer qué método debe ejecutarse y cuándo:

```
touchListener->onTouchBegan = CC_CALLBACK_2(Race::onTouchBegan,
this);
```

En este caso, se indica que debe llamarse al método *onTouchBegan* de la clase *Race*, cuando el *touchListener* recoja que es pulsado. También se podría poner:

```
touchListener->onTouchEnded = CC_CALLBACK_1(Race::anotherMethod,
this);
```

En este caso, se activaría el método *anotherMethod* cuando el usuario deje de pulsar el elemento que tenga asociado *touchListener*. Nótese que se pueden tener, al mismo tiempo, métodos para *onTouchBegan*, *onTouchMoved* y *onTouchEnded*.

Finalmente, tan sólo sería necesario asignar este *listener* a un nodo. Por ejemplo:

```
_eventDispatcher-
>addEventListenerWithSceneGraphPriority(touchListener, _leftArrow);
```

Si se quiere usar un *listener* similar para otro elemento, se puede clonar éste *listener* a la hora de añadirlo al *eventDispatcher*.

```
_eventDispatcher-
>addEventListenerWithSceneGraphPriority(touchListener->clone(),
_rightArrow);
```

5. Pruebas

En este apartado se detallan las pruebas realizadas, con el fin de comprobar que el sistema cumplía los Requisitos Funcionales (*2.1*) y los Requisitos No Funcionales (*2.2*). Para ello, se ejecutan las pruebas desde el punto de vista de un usuario. Esto es, se realizan pruebas de caja negra, verificando que el resultado es el esperado ante una interacción.

La nomenclatura utilizada sigue un patrón similar al de los requisitos, donde PS se corresponde con Pruebas de Sistema, seguido de un identificador.

# Prueba	Descripción	Resultado
PS1	La aplicación se abre correctamente en menos de 5s	
PS2	La aplicación se cierra correctamente al pulsar "*Exit*"	
PS3	La aplicación se ejecuta en modo *portrait*	
PS4	El botón *"Ranking"* muestra la pantalla de *ranking*	
PS5	El botón *"Play"* muestra la pantalla de configuración	
PS6	El botón *"Start"* da comienzo a la carrera con la configuración introducida	
PS7	Los vehículos rivales no tienen la misma textura que el del jugador	
PS8	El menú de pausa se abre correctamente	
PS9	El menú de pausa mantiene el estado de la carrera	
PS10	El mapa se genera en bucle infinito	
PS11	La tasa de *frames per second* se mantiene estable	
PS12	Cuando el jugador pulsa una flecha de dirección, su vehículo se mueve al lado correspondiente	
PS13	El jugador sufre una penalización al colisionar con un obstáculo	
PS14	El jugador sufre una penalización al colisionar con un oponente	
PS15	Los oponentes sufren una penalización al colisionar con un obstáculo	
PS16	El contador de tiempo se reinicia al pasar por línea de meta	
PS17	Al superar la posición de un rival, se actualiza el puesto en carrera	
PS18	Al ser superado por un rival, se actualiza el puesto en carrera	
PS19	Al terminar la carrera, se muestra la vuelta más rápida que ha conseguido el jugador en la misma	
PS20	En el *ranking* se muestran las 3 vueltas más rápidas que se han conseguido	
PS21	Al pulsar el botón de "*Reset*" del *ranking*, éste se limpia